Scalable Parallel Graph Partitioning

Shad Kirmani
Computer Science and Engineering
The Pennsylvania State University
University Park, PA, 16801, USA
sxk5292@cse.psu.edu

Padma Raghavan
Computer Science and Engineering
The Pennsylvania State University
University Park, PA, 16801, USA
raghavan@cse.psu.edu

ABSTRACT

We consider partitioning a graph in parallel using a large number of processors. Parallel multilevel partitioners, such as Pt-Scotch and ParMetis, produce good quality partitions but their performance scales poorly. Coordinate bisection schemes such as those in Zoltan, which can be applied only to graphs with coordinates, scale well but partition quality is often compromised. We seek to address this gap by developing a scalable parallel scheme which imparts coordinates to a graph through a lattice-based multilevel embedding. Partitions are computed with a parallel formulation of a geometric scheme that has been shown to provide provably good cuts on certain classes of graphs. We analyze the parallel complexity of our scheme and we observe speed-ups and cut-sizes on large graphs. Our results indicate that our method is substantially faster than ParMetis and Pt-Scotch for hundreds to thousands of processors, while producing high quality cuts.

1. INTRODUCTION

Graph partitioning has applications in many areas such as circuit design [26] and parallel computing [13]. In this paper, we consider the problem of partitioning a graph. Additionally, we are interested in computing the partition in *parallel*. Such partitioning in parallel is important in many cases where a sequential implementation would pose a serious bottleneck, for example, in a scientific simulation with a large number of processors [6,10,11], where periodically, data and tasks have to be re-distributed in order to re-balance workloads while limiting inter-processor communication.

The graph partitioning problem that we consider concerns an unweighted and undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. We consider sparse graphs with $|V| = N$ and $|E| = M$ such that M is a small constant times N. We seek a partition of the set of vertices V into two disjoint subsets V_1 and V_2 of nearly equal size such that the number of edges connecting a vertex in V_1 to a vertex in V_2 is small. In other words, $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \phi$, and $|V_1| \approx |V_2| \approx \frac{1}{2}|V|$, with an edge separator S of small size, where $S = \{(i,j)|(i,j) \in E$, and $i \in V_1, j \in V_2\}$. Computing an optimal partition, i.e., one with a minimal cut-size $|S|$ and V_1 and V_2 of equal size, is known to be NP-Complete [8]. Consequently, there is an extensive body of heuristics that seek to compute high quality partitionings.

In this paper, we seek a parallel partitioning scheme suitable for some P processors such that the computations at a processor are a small constant multiple of M/P for scalability (costs are typically proportional to the number of edges M) of a sequential algorithm. Ideally, the inter-processor communication costs of the parallel partitioner should be less than the costs of computation and independent of P. However, in practice inter-processor communication cannot be localized to that extent and their costs will grow with P. This is a 'bootstrapping' problem in that the efficiency of any parallel algorithm depends on having a high degree of locality of data on each processor, which is what we are seeking to achieve by computing a graph partition and which is therefore not available for speeding-up a parallel partitioning scheme.

Parallel graph partitioning is inherently challenging because of the the underlying tradeoffs between computing partitions of high quality and limiting inter-processor communication to yield a scalable parallel implementation. Early solutions to this problem were not general purpose schemes and they concerned graphs that have associated coordinates in two or three dimensions such as those related to meshes. Parallel geometric schemes using line or plane separators, including recursive coordinate bisection [1] and Cartesian nested dissection [12] were shown to be scalable at the expense of cut quality. Additionally, a new geometric mesh partitioning scheme using 'spheres' was shown to be provably good for certain classes of well-shaped finite-element meshes [9, 24]. A sequential Matlab implementation showed that underlying algorithms produced high quality cuts and may yield an effective parallel formulation [9]. More recently, multilevel schemes for parallel graph partitioning including ParMetis [19, 20] and Pt-Scotch [25] have been shown to

This research was supported in part by the National Science Foundation through awards 0821527, 0963839, and 1017882.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC'13 November 17 - 21 2013 Denver, Colorado USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.

have efficient parallel performance on modest numbers of processors while producing high quality cuts; however, their performance suffers at higher processor counts.

In this paper, we seek a new general purpose graph partitioning scheme that can incorporate the best attributes of multilevel and geometric methods to compute high quality partitions in parallel on larger processor counts. Our main contribution is a fast parallel multilevel graph embedding scheme to impart coordinates to an arbitrary graph. These coordinates are then used to perform parallel geometric partitioning. However, instead of using line and plane cuts as in recursive coordinate bisection and Cartesian nested dissection, we develop a parallel implementation of the geometric mesh partitioning schemes of Gilbert et. al. [9, 24].

The rest of this paper is organized as follows. In Section 2, we provide background and a brief overview of related work. In Section 3, we develop and analyze our parallel partitioning scheme with a focus on our key contributions including a fixed lattice multilevel parallel graph embedding scheme and a parallel form of the geometric mesh partitioning scheme of Gilbert et al. [9]. In Section 4, we evaluate the performance of our partitioning scheme and include comparisons with established schemes such as Pt-Scotch [25], ParMetis [19, 20] and recursive coordinate bisection(RCB) [1, 2]. We summarize our contributions and remark on directions for future work in Section 5.

2. BACKGROUND AND RELATED WORK

We leverage the extensive body of prior work on partitioning schemes to develop our new parallel partitioner. Below, we provide a brief summary of prior results that are directly related to the scheme that we will develop and analyze in this paper.

Parallel Graph Partitioning. There is a very large body of earlier work on graph partitioning [6], including spectral, multilevel and geometric schemes and their hybrids; our work builds upon multilevel and geometric schemes.

Our proposed work leverages the techniques in parallel multilevel graph partitioning algorithms such as Chaco [14, 15], ParMetis [19,20] and Pt-Scotch [25]. In these multilevel schemes a graph is coarsened repeatedly to get sequence of coarsened, smaller graphs with an approximate halving of the number of vertices at each step. The coarsest graph of very small size is then partitioned. Each coarse graph is then uncoarsened to the next level, the partition at the coarse level is projected to the next finer graph and refined [7, 21]. This process is repeated until a partition is obtained in the original graph.

Parallel geometric partitioning schemes assume the vertices of the graph have associated coordinates in two or three dimensions which are then utilized to compute line or plane separators [1, 2, 12] The Zoltan project [2] shows that such coordinate bisection can be computed very fast in parallel using large numbers of processors. However, the quality of cuts are typically worse than those produced by parallel multilevel schemes such as Pt-Scotch and ParMetis. An alternate geometric mesh partitioning scheme [9,24] has been shown to compute provably good partitions under certain assumptions regarding the mesh. This scheme constructs 'sphere' separators by projecting the vertices of the graph onto the surface of a sphere in a higher dimension, computing a centerpoint, selecting a random great circle through the centerpoint and projecting this great circle back to the original coordinate space to define a partition. Typically, the partition with the smallest edge separator size is selected after computing 20-30 different partitions. A serial Matlab implementation is also provided by the authors in the appendix of the paper [9, 24]. Although this scheme has not been parallelized, it is expected to yield good parallel formulations based on the results related to parallel coordinate bisection.

More recently, Holtgrewe et. al. [16] have developed KaPPa to utilize coordinate information for the vertices of the graph to speedup parallel multilevel graph partitioning. Coordinate information is used to compute an initial assignment of vertices to processors followed by multilevel partitioning. Additionally, partition refinement is performed by processor pairs on a shared edgecut.

A diffusion based multilevel scheme for computing high quality graph partitions has been developed by Meyerhenke et. al. [22,23]. The scheme called DIBAP [22] is able to find very high quality partitions. However, the parallel implementations is significantly slower than ParMetis [22].

Force Based Multilevel Graph Embedding. In order to develop our parallel partitioning scheme, we leverage prior work (unrelated to graph partitioning), that concern graph drawing. Specifically, we adapt a force based method similar to the one proposed by Hu [17]. A similar work using the Fast Multipole Method for multicores and GPUs has been also been done [27]. We develop a much simpler version of the force based algorithm described below.

The embedding scheme involves N-body force calculations where each vertex exerts an attractive force on its neighbor vertices while it is repelled by *all* other vertices in the graph. The magnitude of the attractive F_a and repulsive F_r forces on a vertex i of $G = (V, E)$ are $F_a(i) = \sum_{(i,j)\in E} \frac{\|c_i - c_j\|^2}{K}$ and $F_r(i) = -\sum_{j=1, i\neq j}^{N} \frac{CK^2}{\|c_i - c_j\|}$ where c_i and c_j represent coordinates of vertices i and j, $\|c_i - c_j\|$ represents the distance between vertices i and j. C and K are suitable 'twiddle-factors' known to work well in practice [17]. The Barnes-Hut approach is used to approximate repulsive forces to reduce the cost of force calculations to $\mathcal{O}(n\ log\ n)$ instead of $\mathcal{O}(n^2)$ for the naive approach. At every iteration, each vertex is moved a small distance in the direction of the net force on that vertex, for convergence after a certain number of iterations. In a sequential multilevel form [17], the graph is repeatedly coarsened much as in multilevel schemes for graph partitioning. The coarsest graph is then embedded using the force based scheme. It is then successively uncoarsened with vertices at each finer level inheriting the embedding from the coarser level followed by a few iterations of the force based scheme to smooth the embedding, until an embedding is obtained for the original graph.

3. SCALAPART: PARALLEL MULTILEVEL EMBEDDED GRAPH PARTITIONING

In this section, we develop and analyze our parallel multilevel embedded graph partitioner that we henceforth refer to as ScalaPart (SP).

We begin by motivating the development of ScalaPart in terms of differences and similarities to related prior work in Section 2. Recall that multilevel parallel partitioners such as ParMetis [19, 20] and Pt-Scotch [25] produce high quality partitions but their fixed problem speed-ups suffer from the costs related to un-coarsening and refinement. In ScalaPart, we coarsen graphs in the same manner as in ParMetis. However we change the successive un-coarsening and refinement to seek a new scheme with with improved parallel performance. More specifically, we impart coordinates to the coarsest graph using force based embedding [17] and we refine the embedding in a parallel multilevel form. We develop a fixed lattice parallel embedding scheme for fast smoothing of inherited coordinates from a coarser level. At the end of the multilevel process, the original graph has coordinates and we partition it using a parallel formulation of the geometric mesh scheme [9]. We thus seek to gain the benefits of a multilevel approach for effective graph contraction, general applicability to arbitrary graphs that may not have coordinate information, improvements in cut quality from mesh partitioning, and the scalability of parallel geometric partitioning.

Consider a graph $G = (V, E)$ distributed on some P processors, represented as $G(P)$. Below is a high level overview of our parallel graph partitioning algorithm in terms of its three main steps. Details of each step are provided after the this overview.

- **Coarsening.** The graph is read in by P processors in approximately equal sized chunks. The graph is coarsened using the heavy-edge matching as in ParMetis [19, 20] to yield a sequence of graphs where each graph is approximately one half the size of the preceding one. One minor adaptation is that we only retain every other graph to get a sequence with sizes that decrease approximately by a fourth. We thus obtain $G^0(P^0), G^1(P^1), \cdots G^k(P^k)$, where $G^0(P^0) = G(P) = (V, E)$ and $G^i(P^i) = (V^i, E^i)$, $i = 1, \cdots k$, such that $|V^i| \approx \frac{1}{4}|V^{i-1}|$ and $P^i \approx \frac{1}{4}P^{i-1}$; the number of levels k is selected so that V^k is suitably small.

- **Multilevel Fixed Lattice Parallel Graph Embedding.** The coarsest graph, $G^k(P^k)$ is embedded using P^k processors,to yield $G^k(P^k, C^k)$, i.e., $G^k(P^k)$ with coordinates C^k for its vertices, through a fixed lattice parallel force based embedding scheme that we develop below as our main contribution. The multilevel embedding proceeds by using the fixed lattice scheme to smooth $G^i(P^i, C^i), i = k-1, k-2, \cdots 0$ after each vertex j in V^i inherits a scaled form of the embedding (coordinates) from the vertex (super-vertex) j in $G^{i+1}(P^{i+1}, C^{i+1})$.

- **Parallel Geometric Partitioning and Refinement.** The original graph $G = G^0(P^0)$, now with coordinates as $G(P, C) = G^0(P^0, C^0)$ is partitioned in parallel using a modified form the geometric mesh partitioning scheme [9]. Our modifications seek to manage the trade-off between cut quality and parallel performance, including the identification of a strip around a computed partition to which Fiduccia-Mattheyses refinement [7] can be applied; such refinement is known to reduce the size of the edge separator.

Fixed Lattice Parallel Graph Embedding. Our main contribution is a fixed lattice parallel graph embedding that can be computed effectively in parallel. Consider embedding a graph $G = (V, E)$ using some P processors arranged in a two dimensional grid. Assume the following:

- There is an initial assignment of coordinates to vertices in V, and that all vertices lie within a box B with corners given by $[x^l, y^b], [x^r, y^b], [x^l, y^t], [x^r, y^t]$. At the coarsest level, vertices are initially assigned coordinates randomly; this embedding is refined using the force based scheme. At all other levels, the coordinate information is inherited from the preceding level as explained under "Multilevel Projection and Smoothing" in the following page.

- P processors are arranged in a $\sqrt{P} \times \sqrt{P}$ processor grid with $P_{i,j}$ denoting the i, j-th processor for $i, j \in \{0, 1, \cdots, \sqrt{P} - 1\}$. Likewise, the bounding box B is also viewed in the form of a $\sqrt{P} \times \sqrt{P}$ lattice with $B_{i,j}$ representing the i, j-th sub-domain.

- $G_{i,j} = (V_{i,j}, E_{i,j}, C_{i,j})$ denotes the subgraph whose vertices have coordinates in $B_{i,j}$; it is mapped to processor $P_{i,j}$. Define its set of boundary vertices as $\tilde{V}_{i,j} = \{p : p \in V_{i,j}, (p, q) \in E_{i,j}, \text{and}, q \notin V_{i,j}\}$; these are vertices in $V_{i,j}$ connected by an edge to a vertices in a different sub-domain assigned to a different processor. Define as *ghost* vertices $\hat{V}_{i,j} = \{q : p \in V_{i,j}, (p, q) \in E_{i,j}, \text{and}, q \notin V_{i,j}\}$, i.e., those vertices connected to the boundary vertices in $V_{i,j}$.

- Each vertex in $\hat{V}_{i,j}$ is assigned coordinates so the edges between $\tilde{V}_{i,j}$ and $\hat{V}_{i,j}$ have endpoints in one of the four neighboring boxes $B_{i\pm1,j\pm1}$, i.e., at shortest L_1 distance from the sub-domain to which it belongs. For example, if a vertex $q \in \hat{V}_{i,j}$ belongs to $G_{i+4,j}$ it would have coordinates in box $B_{i+1,j}$.

At each iteration, each processor $P_{i,j}$ is responsible for the force calculations on the particles it owns. Attractive forces are calculated using the standard expression (see Section 2) with distance calculations using the coordinate information of vertices. However, long range repulsive forces are approximated using the domain lattice in order to limit inter-processor communication and to reduce the computation per particle. Our approach can be viewed as fixed lattice Barnes-Hut type approximation. Each domain lattice $B_{i,j}$ is associated with a special vertex $\beta_{i,j}$ with mass $\mu_{i,j}$ corresponding to all vertices in $B_{i,j}$ and located at their center of mass at $\phi_{i,j}$. Each processor independently calculates $\phi_{i,j}$ and $\mu_{i,j}$ followed by a reduction across processors to aggregate this information. Next $P_{i,j}$ calculates the repulsive force on $\beta_{i,j}$ from all other special β vertices as follows.

$$F_r(\beta_{i,j}) = - \sum_{\substack{q,r,q\neq i,r\neq j,\\ q,r\in\{0,\cdots\sqrt{P}-1\}}} \frac{CK^2}{\|\phi_{i,j} - \phi_{q,r}\|} \times \mu_{i,j} \times \mu_{q,r} \quad (1)$$

All vertices in $V_{i,j}$ inherit the repulsive force on $\beta_{i,j}$. Moreover, each vertex in $V_{i,j}$ is also repelled by its special vertex $\beta_{i,j}$. Hence, to calculate the long range repulsive forces on a vertex $q \in V_{i,j}$ with coordinates c_q and mass μ_q, we use the modified equation below.

$$F_r(q) = F_r(\beta_{i,j}) - \frac{CK^2}{\|c_q - \phi_{i,j}\|} \times \mu_{i,j} \times \mu_q \qquad (2)$$

Following the force calculation, vertices in $V_{i,j}$ are moved under the influence of the force but vertices in $\hat{V}_{i,j}$ are left fixed. See Figure 1 for an example graph on a 3×3 domain lattice and processor grid.

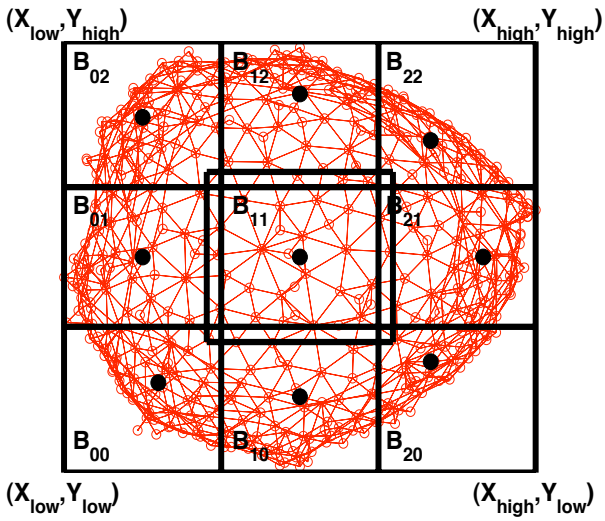


Figure 1: Example graph using a 3×3 processor grid and domain lattice; a large dot represents the special vertex β used in the approximation to the repulsive forces.

To reduce communication, coordinate information of vertices corresponding to edges that cross two processors, and the domain lattice special vertex data are collected per iteration using only *local nearest neighbor* communication on the processor grid. Coordinate information for edges that cross beyond domain blocks assigned to nearest neighbors on the processor grid and domain lattice special vertex data beyond those of nearest neighbors on the processor grid are *not* collected at each iteration but instead collected once after a block of several iterations. Global communication, including an all gather and a reduction are performed only once per block. Iterations within a block thus act on stale data For block sizes comprising 2-8 iterations, there was no observable change in the quality of the embeddings while global communication costs were correspondingly reduced. Only these require global communication including an all gather and a reduction operation. Intermediate iterations thus act on stale data but global communication is reduced.

Multilevel Projection and Smoothing. Recall that our coarsening yielded a sequence of graphs $G^0(P^0), G^1(P^1), \cdots G^k(P^k)$, where $G^0(P^0) = G(P) = (V, E)$ and $G^i(P^i) = (V^i, E^i), i = 1, \cdots k$, such that $|V^i| \approx \frac{1}{4}|V^{i-1}|$ and $P^i \approx \frac{1}{4}P^{i-1}$. The coarsest graph $G^k(P^k)$ is embedded using P^k processors, to yield $G^k(P^k, C^k)$, i.e., $G^k(P^k)$ with coordinates C^k for its vertices in a box B^k using the fixed lattice scheme described above. We expect P^k to be a small number such as 4 or 8 and the number of vertices of G^k is also assumed to be small, in the hundreds or few thousands.

Following the coarsest graph embedding, we apply a recursive coordinate bisection scheme such as the one in Zoltan [2] to map vertices of G^k embedded in B^k to some $p^k \times q^k$ processor grid with a total of P^k processors. This mapping determines the lattice sub-domain ${B^k}_{i,j}$ and corresponding embedded subgraph ${G^k}_{i,j} = ({V^k}_{i,j}, {E^k}_{i,j}, {C^k}_{i,j})$ for processor (i, j) in the processor grid. This information is used to project a mapping of the refined graph G^{k-1}, to P^{k-1} processors arranged in a $2p \times 2q$ grid, and its initial embedding onto a domain with a bounding box B^{k-1} as follows.

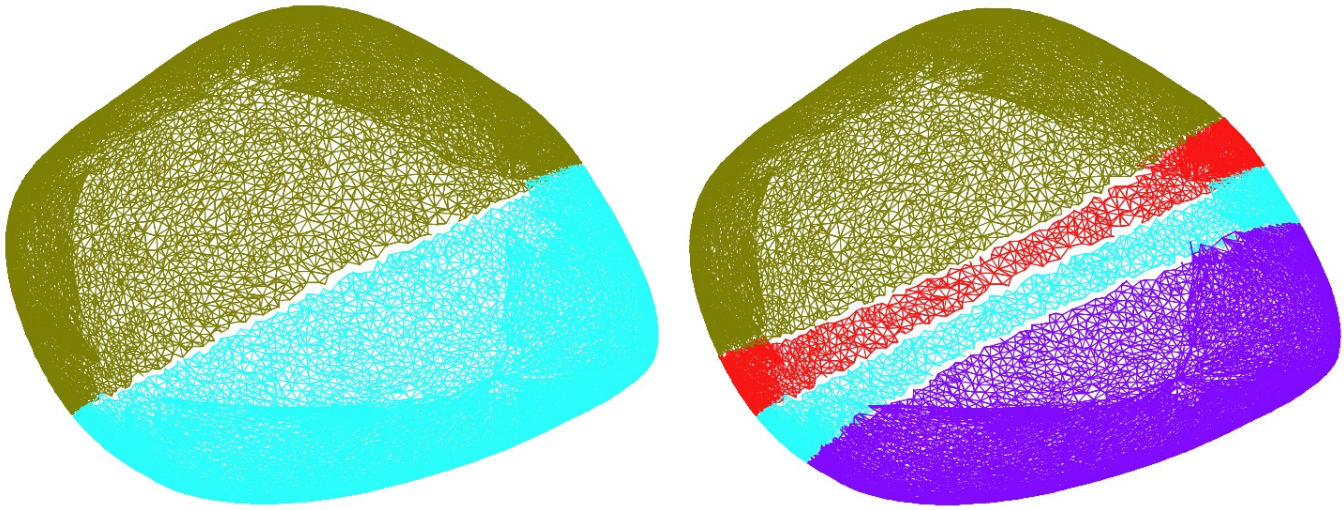
- Observe that a vertex in G^k represents approximately four vertices in G^{k-1}. The bounding box B^k is scaled by a factor of two in each dimension to get a bounding B^{k-1} that is roughly four times as large.
- Each box in the lattice sub-domain ${B^k}_{i,j}$ and the coordinates of vertices in G^k are scaled by factor of two in each dimension.
- A set of vertices in G^{k-1} are placed with small translations about the center point given by the coordinates of their coarse graph vertex in G^k.
- Lattice sub-domains and processor assignments in G^{k-1} are obtained by a 2×2 splitting of each ${B^k}_{i,j}$ in G^k and distribution of related data using nearest neighbor communication on the processor grid.

Following the *projection* as described above, G^{k-1} is *smoothed* using a few iterations of the fixed lattice embedding scheme. Observe that successive applications of this process will leave in the original graph, now with an embedding, distributed across all processors as $G(P, C) = G^0(P^0, C^0)$.

Parallel Geometric Mesh Partitioning and Partition Refinement We partition the original graph $G(P, C) = G^0(P^0, C^0)$ on P processors using a natural parallel formulation of the geometric mesh partitioning scheme [9]. Key elements include the following.

- We use sampling across processors to calculate the centerpoint fast.
- Multiple great circles (and corresponding separators) are computed redundantly on each processor.
- Each processor computes its contribution to the measure of cut quality for all separators, before a reduction involving all processors to select the best cut.

Our parallel partitioner only calculates sphere separators and avoids the eigenvector calculation needed for a line separator in the interests of parallel scalability. To refine the partition we select circles neighboring the separating circle to identify a strip to which we may apply a Fiduccia-Mattheyses refinement [7, 21]. Typically the strip contains a small multiple of the number of vertices in the edge separator. Consequently the costs of this step are negligible.



(a) Edge separator.

(b) Strip around the edge separator.

Figure 2: Example of a strip used to refine the edge separator for delaunay_n16. This strip contains as many vertices as 5.6 times the number of edges in the separator.

Our refinement on a strip is similar to the band graph approach in Pt-Scotch [25] with the following difference. Instead of selecting the band using vertices within a certain number of hops for an end point of an edge in the separator, we select a strip using coordinate information. Figure 2 illustrates our approach.

3.1 Parallel Complexity of ScalaPart

The parallel complexity of ScalaPart differs from those of other multilevel methods such as Pt-Scotch and ParMetis, only in the costs of inter-processor communication for the multilevel fixed lattice embedding and the geometric partitioning of the embedded graph.

Consider the communication required for parallel multilevel embedding with *projection* from a coarse graph to the finer graph at next level and *smoothing* using a fixed lattice embedding. Recall that inter-processor communication for projection involves only nearest neighbor messaging on the processor grid. Costs are independent of the number of processors and thus scale well. However, each smoothing step requires higher amounts of communication as shown below for some G^i, where $i = 1, \cdots, k$, with $G^0 = G$, $P^0 = P$ and $P^i \approx \frac{1}{4}P^{i-1}$. We use t_s and t_w to denote latency and per-word costs of inter processor communication.

- Each iteration requires local nearest neighbor communication of boundary points for costs of $c_0(t_s + t_w\sqrt{\frac{|V^i|}{P^i}})$ for G^i where c_0 is a small constant representing the total number of iterations.
- A few global all gather operations for G^i using P^i with costs that grow as $(t_s \log P_i + t_w \tilde{n})$. Here, $\hat{n}$ is the total number of edges that span non nearest neighbor blocks on the domain lattice and thus processors that are not nearest neighbors on the processor grid; $\tilde{n}$ is typically much smaller than the number of boundary points for G^i given by $\sqrt{\frac{|V^i|}{P^i}}$.
- A few reductions for G^i using P^i with costs that grow as $(t_s \log P_i + t_w P_i \log P_i)$.

We need to sum these costs across all levels G^i, $i = 1, \cdots, k$, with $G^0 = G$, $P^0 = P$ and $P^i \approx \frac{1}{4}P^{i-1}$. With the quadrupling of P_i across successive levels, the total number of levels k is proportional to $\log_4 P$. Let $\tilde{N}$ denote a value that is much smaller than $\sqrt{\frac{N}{P}}$. Now the total costs of communication will grow as:

$$t_s(\log P)^2 + t_w P(\log P)^2 + t_w \tilde{N} \log P + t_w\sqrt{\frac{N}{P}}.$$

We expect that the costs related to message latency of the form $t_s(\log P)^2$ will be dominant and thus impacting parallel performance. However, these are still relatively small, similar to those for pivot computation in a parallel quicksort implementations.

For the parallel geometric partitioning of G using P processors, the communication costs are far lower than those for smoothing. They correspond to 3 reductions with short messages of sizes that are effectively a constant c (corresponding to the number of great circle separators). Total costs for partitioning are low at $3(t_s + t_w c \log P)$.

4. EVALUATION OF PARALLEL PERFORMANCE AND PARTITION QUALITY

In this section, we measure and report on the performance of our implementation of ScalaPart (SP) relative to those of other well-established and widely used parallel partitioning schemes.

Evaluation Methodology and Measures. Parallel partitioning schemes that we consider for a comparative evaluation include the following.

- ParMetis [19, 20] developed by Karypis et al.

- Pt-Scotch [25] developed by Chevalier and Pellegrini.
- Recursive coordinate bisection (RCB) from Zoltan [2].
- The sequential Matlab implementation of the geometric mesh partitioner [9] developed by Gilbert et al. that we denote by G30 when the best cut is selected out of 30 tries including 22 great circle separators with 2 centerpoints and 7 line separators. We also used two variants, G7 with the best cut out of 7 tries including 5 great circles with 1 centerpoint and 2 line separators and G7-NL denoting G7 without the line separator.

We denote our implementation of ScalaPart by SP and we refer to the parallel geometric partitioner within ScalaPart by SP-PG7-NL to indicate the fact that it is a parallel formulation of G7-NL with strip-refinement using Fiduccia-Mattheyses [7]. We were not able to compare ScalaPart with the parallel partitioner KaPPa [16] because a software implementation of KaPPa is not available. However, we would like to note that "ParMetis is an order of magnitude faster" as observed by the authors of KaPPa [16] and it can be used only on graphs that already have coordinate information.

We use a test suite of large graphs from the UFL sparse Matrix collection [5] as shown in Table 1. In these graphs, the number of vertices N range from 1-21 million, and the number of edges M range from 5-100 million. RCB and the geometric mesh partitioners G30, G7, and G7-NL require a graph with coordinates. We provide such coordinates using the force-based graph drawing code in Mathematica developed by Hu [18] which typically provides very high quality embeddings.

Table 1: Test suite of graphs.

	N(10^6)	M(10^6)
ecology1	1	4.99
ecology2	0.99	4.99
delaunay_n20	1.05	6.29
G3_circuit	1.58	7.66
kkt_power	2.06	12.77
hugetrace-00000	4.59	13.76
delaunay_n23	8.39	50.33
delaunay_n24	16.77	100.66
hugebubbles-00020	21.20	63.58

Our experiments were performed using $P = 1, 2, \cdots, 1024$ processors of a cluster with 128 nodes having two quad core Nehalem processors at 2.66GHz and a QDR Infiniband interconnect.

We used as the measure of *parallel performance* the execution time on $1 - 1,024$ processors. We report on observed execution times to partition a graph into two parts by computing a single edge separator. As the measure of *partition quality* we use the cut-size, i.e., the size of the edge separator. The cut-size varies with P the number of processors for all methods with the exception of RCB; we show the observed range of cut-sizes for all methods except RCB.

Table 2: Comparison of partition quality across geometric schemes. G30 is a sequential geometric mesh partitioning with 30 tries, G7 with 7 tries while G7-NL has 7 tries and no line separator. RCB denotes recursive coordinate bisection. Cut-sizes are shown relative to G30 set at 1. Avg SP and Best SP respectively show the average and best cut-sizes from SP across processors in the range 1-1,024. Here hugetrace denotes hugetrace-00000, and hugebubbles denotes hugebubbles-00020.

	Relative cut-sizes of geometric methods				
	G7	G7-NL	RCB	Avg SP	Best SP
ecology1	1.00	1.01	1.06	0.92	0.80
ecology2	0.99	1.00	0.99	0.91	0.80
delaunay_n20	0.96	1.03	1.16	0.82	0.51
G3_circuit	1.00	1.01	1.03	0.70	0.59
kkt_power	1.46	1.45	1.51	0.92	0.51
hugetrace	1.03	1.03	1.09	0.85	0.77
delaunay_n23	1.08	1.29	1.27	0.78	0.72
delaunay_n24	0.98	1.07	1.24	0.86	0.74
hugebubbles	1.10	1.10	1.15	0.86	0.76
Geom. Mean	1.06	1.10	1.16	0.84	0.68

Evaluation of Partition Quality. We first compare cut-sizes across all partitioners that utilize coordinate information, namely the sequential mesh partitioning [9] variants G30, G7, and G7-NL by Gilbert et al., the parallel recursive coordinate bisection scheme RCB from Zoltan [2], and our parallel formulation of mesh partitioning in ScalaPart, namely SP-PG7-NL. Table 2 shows cut-sizes relative to those produced by G30 set at 1; values smaller than 1 indicate relative improvements while values greater than 1 indicate relative degradation. Observe that RCB performs 16% worse than G30 on average while G7-NL performs worse on average by 10%. ScalaPart uses a parallel form of G7-NL; not surprisingly, its worst performance is comparable to G7-NL at approximately 6% worse than G30. However, its best cuts are on average 32% better than G30. The improvements relative to G30 and G7-NL can be attributed to reductions in cut-size in ScalaPart from the strip-refinement using Fiduccia-Mattheyses [7].

Table 3 shows the range of cut-sizes for Pt-Scotch, ParMetis, ScalaPart (SP), G30 and RCB. Observe that in most cases, the best (smallest) cut-size of SP is better than the best among all methods. Cutsizes are somewhat higher for Par-Metis. On average, relative to the best cut-size of Pt-Scotch set at 1, the worst cut-size of Pt-Scotch is 42% larger, while the best cut-size for ScalaPart is better by 6% with the worst being 47% larger. On average cut-sizes for RCB and geometric mesh partitioning are worse by 61% and 39% respectively, while ParMetis cuts are 10% to 67% larger. Thus for this test suite, partitions produced by ScalaPart compare favorably with those produced by Pt-Scotch and are generally better than those for ParMetis, G30 and RCB. We conjecture that the cut quality of PartMetis reflects a trade-off in favor of faster coarsening and refinement. Additionally, ScalaPart considers several geometric separators of the original (fine) graph and selects the best from among them; this approach generally results in higher quality par-

Table 3: Test graph collection with best and worst cut-sizes for all methods. The best and worst values for each graph across all methods are shown in **bold** and underline respectively.

	Pt-Scotch	ParMetis	ScalaPart	G30	RCB
ecology1	**1,094** - 1,500	1,229 - 1,446	1,115 - 1,436	1,394	1,473
ecology2	1,144 - 1,377	1,236 - 1,515	**1,111** - 1,555	1,388	1,380
delaunay_n20	1,920 - 2,091	2,085 - 2,494	**1,339** - 2,708	2,603	3,018
G3_circuit	1,205 - 1,592	1,433 - 2,068	**1,199** - 1,776	2,018	2,069
kkt_power	19,877 - 76,267	20,930 - 106,390	**15,998** - 40,521	31,503	47,563
hugetrace-00000	**770** - 937	786 - 1,117	780 - 1,063	1,018	1,112
delaunay_n23	5,521 - 7,674	5,959 - 8,248	**5,466** - 6,841	7,578	9,639
delaunay_n24	7,884 - 9,544	8,775 - 12,086	**7,835** - 12,695	10,643	13,176
hugebubbles-00020	**1,474** - 1,847	1,656 - 2,170	1,563 - 2,278	2,059	2,363
Geometric Mean	1.00 - 1.42	1.10 - 1.67	0.94 - 1.47	1.39	1.61

titions.

Parallel Performance. Figure 3 shows total execution times on 1-1,024 processors *over all test graphs* in Table 1 for all methods including ScalaPart, Pt-Scotch, ParMetis and RCB. For RCB we provide coordinates using the graph drawing code developed by Hu [17]; times for the embedding are not included. As expected, on small numbers of processors ScalaPart is substantially slower because of the larger number iterations needed to obtain an embedding. However, for higher values of P, starting at around 64, ScalaPart becomes competitive and outperforms both Pt-Scotch and ParMetis on 256 through 1,024 cores. This is despite some performance degradation for the smaller graphs on these higher processor counts of 256-1024 caused by the higher costs of communication relative to those for computation (this effect is less pronounced for the larger graphs, as shown by Figure 9). On 1,024 processors ParMetis requires only 23.75% of the time required by Pt-Scotch while ScalaPart is even faster requiring only 6.17% of the time required by Pt-Scotch. Observe also that the execution times for ScalaPart approach those for RCB at 1,024 processors.

Figure 4 shows a detailed comparison of the total execution times over all test graphs for RCB and SP-PG7-NL, i.e., ScalaPart exclusive of coarsening and embedding and including only parallel partitioning and refinement. This comparison reflects actual use cases when partitioning in parallel is needed for graphs that already have coordinates such as those from finite-difference and finite-element applications. Starting at 128 processors, our method is faster than RCB while providing significantly better cuts.

We consider the comparisons in Figure 3 for computing a single cut as a tough test of the viability of ScalaPart because the considerable costs of computing an embedding are not amortized over multiple cuts. The incremental costs of computing a cut in ScalaPart are quite competitive as shown in Figure 4.

Figures 5 and 6 show execution times for two representative graphs from our test suite. On 1,024 processors for G3_circuit, relative to Pt-Scotch ParMetis is 77% faster while ScalaPart is 97% faster. For hugebubbles-00020 on 1,024 processors, the differences are slightly less pronounced with ParMetis 48% faster and ScalaPart is 91% faster than Pt-Scotch. Equivalently, as shown in Table 4, for G3_circuit, relative to Pt-Scotch, ParMetis shows a speed-up of 4.28 while ScalaPart shows a speed-up of 32.21 on 1,024 processors. Similarly, speed-ups for hugebubbles-00020 on 1,024 processors are 1.92 for ParMetis and 10.75 for ScalaPart relative to Pt-Scotch.

We consider the parallel performance of ScalaPart in detail in Figures 7 and 8. Figure 7 shows the times for coarsening, embedding and partitioning as a fraction of the total time over all test graphs. We observe that times for embedding are by far the largest fraction of the time in ScalaPart. Figure 8 shows that the relative fraction of time spent in communication for the embedding component starts to increase for larger numbers of processors. This increase in communication is in large part due to the fact that in our preliminary implementation we have used more global communication operations including all gather and all reduce functions than in the algorithm as described in Section 3 and analyzed in Section 3.1. However, despite this shortcoming, in going from 256 to 1,024 processors, the communication time as a fraction of the total embedding time does not increase much because relatively fewer iterations are required at high processor counts for smoothing.

In Figure 9, we show the parallel execution times on 16 through 1,024 processors for Pt-Scotch, ParMetis and ScalaPart for the four largest graphs, namely hugetrace-00000, delaunay_n23, delaunay_n24 and hugebubbles-00020, and the average time across these four graphs. We observe that although ScalaPart is significantly slower at 16 processors than Pt-Scotch while ParMetis is the fastest, the situation is quite the opposite at 1,024 processors. At 1,024 processors, relative to Pt-Scotch, ScalaPart shows a speed-up of 14.37 which is substantially higher that the speed-up of ParMetis at 3.42 (see summary in Table 4).

Table 4 summarizes speedup at 1,024 processors of ParMetis, RCB, ScalaPart and the parallel partitioning component of ScalaPart(SP-PG7-NL) relative to Pt-Scotch. Over all graphs, the speed-up of ParMetis is 4.21 while ScalaPart is even faster with a speed-up of 16.23 relative to Pt-Scotch. RCB shows a speed-up of 25.69 relative to Pt-Scotch. More significantly, when coordinates are available, RCB should be

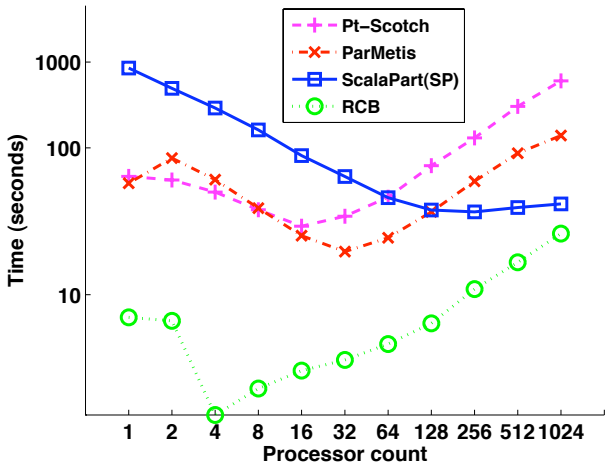


Figure 3: Total execution times over all 9 graphs.

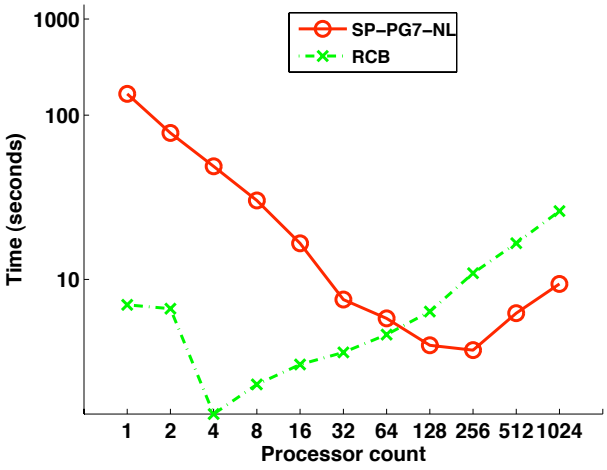


Figure 4: Execution times for RCB and SP-PG7-NL, i.e., ScalaPart times excluding coarsening and embedding times.

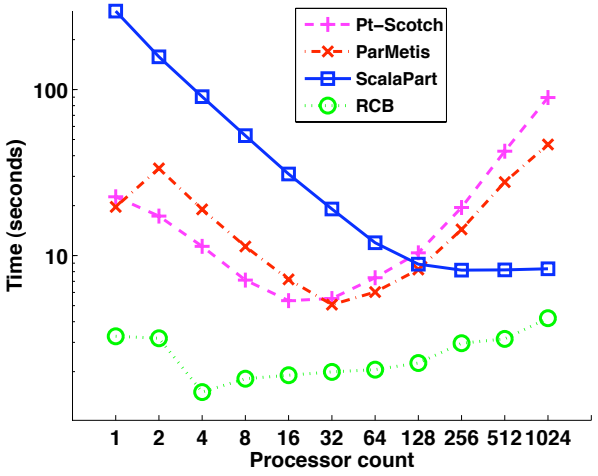


Figure 5: Execution time for hugebubbles-00020.

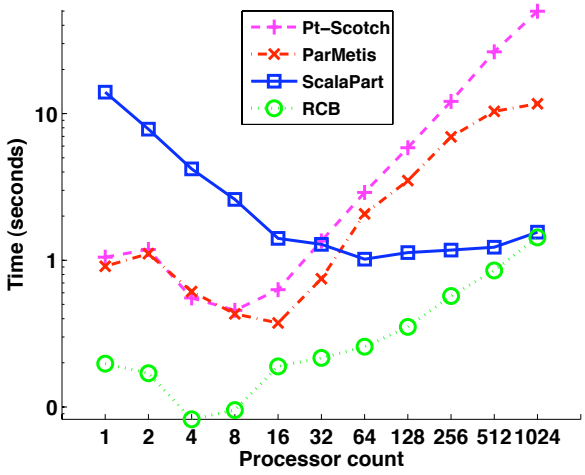


Figure 6: Execution time for G3_circuit.

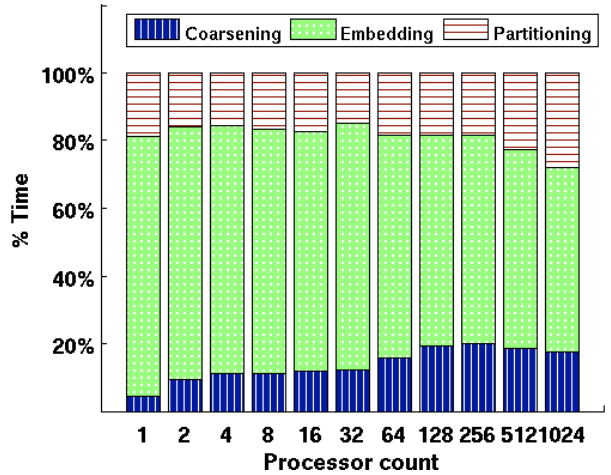


Figure 7: ScalaPart component times.

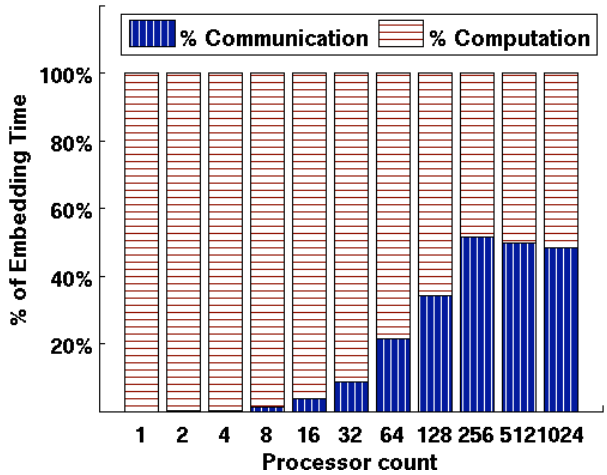


Figure 8: Embedding time composition.

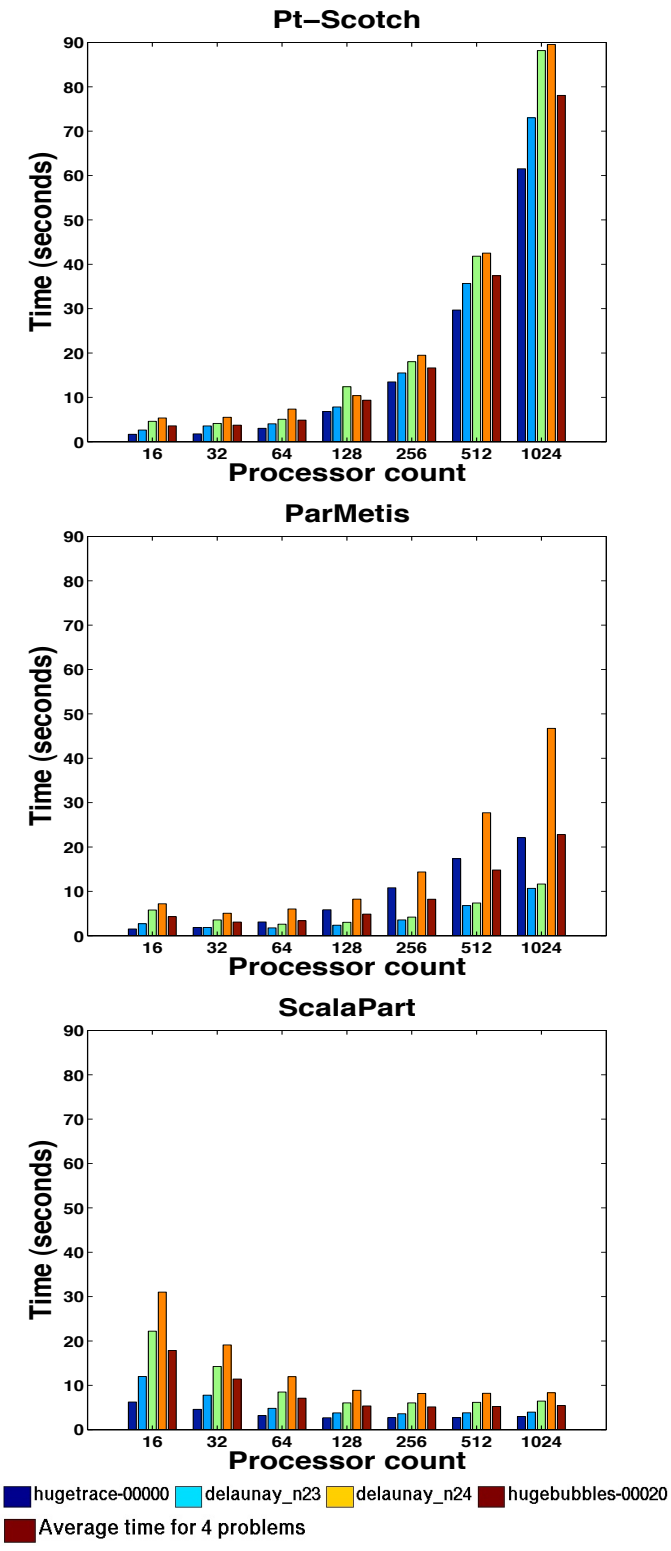


Figure 9: Parallel execution times for Pt-Scotch, ParMetis and ScalaPart for the 4 largest graphs; average times across the four graphs are shown as the fifth bar in each group.

compared to the parallel partitioning component of Scala-Part, namely SP-PG7-NL, which shows a speed-up of 57.92 relative to Pt-Scotch. These results are promising especially given that the quality of partitions computed by ScalaPart are often considerably better than those computed by RCB.

Table 4: Speed-ups at 1,024 processors relative to Pt-Scotch set at 1.

	ParMetis	RCB	ScalaPart	SP-PG7-NL
G3_circuit	4.28	34.92	32.21	74.52
hugebubbles	1.92	21.37	10.75	75.24
All Graphs	4.21	25.69	16.23	57.92
Large 4 graphs	3.42	22.64	14.37	77.48

5. CONCLUSIONS

In this paper, we have developed and analyzed a parallel partitioning scheme that generalizes geometric partitioning by imparting coordinates to arbitrary graphs while computing high quality cuts similar to those computed by Pt-Scotch and often better than cuts computed by ParMetis and RCB. Additionally, our method can be significantly faster than Pt-Scotch for larger numbers of processors. If our method is considered for use of dynamic re-partitioning of graphs with known coordinates, similar to the manner in which RCB can be used, times for partitioning with our scheme (exclusive of embedding) can be significantly faster than those for RCB for larger numbers of processors. We thus add to the choice of parallel partitioning methods that can be used for producing high quality partitions on hundreds to thousands of processors.

We conjecture that the execution times for our method may be improved by using largely local communications as described in Section 3 through a more optimized implementation. Embedding times may also potentially decrease if sampled spectral distance embedding [3] schemes can be combined with our current approach. Additionally, we expect to study in greater detail and for different types of graphs, the relationship between embedding time, quality and partition quality.

6. ACKNOWLEDGMENTS

The authors acknowledge the work by Anirban Chatterjee in an initial study of sequential embedding and its role in graph partitioning as part of his Ph.D. thesis [4].

7. REFERENCES

[1] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, 36(5):570–580, 1987.

[2] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide. Technical Report SAND2007-4748W, Sandia National Laboratories, Albuquerque, N.M., 2007.

[3] A. Çivril, M. Magdon-Ismail, and E. Bocek-Rivele. Ssde: Fast graph drawing using sampled spectral distance embedding. In M. Kaufmann and D. Wagner,

editors, *Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 30–41. Springer Berlin Heidelberg, 2007.

[4] A. Chatterjee. *Exploiting Sparsity, Structure, and Geometry for Knowledge Discovery*. PhD thesis, Pennsylvania State University, 2011.

[5] T. A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 92, 1994.

[6] K. D. Devine, E. G. Boman, and G. Karypis. *6. Partitioning and Load Balancing for Emerging Parallel Applications and Architectures*, chapter 6, pages 99–126. SIAM, 2007.

[7] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[8] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[9] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. In *In Proceedings of International Parallel Processing Symposium*, pages 418–427, 1995.

[10] F. Gioachin, P. Jetley, C. L. Mendes, L. V. Kale, and T. R. Quinn. Toward Petascale Cosmological Simulations with ChaNGa. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[11] A. Y. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the Barnes-Hut method for N-body simulations. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 439–448, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[12] M. T. Heath and P. Raghavan. A Cartesian Parallel Nested Dissection Algorithm. *SIAM J. Matrix Anal. Appl.*, 16(1):235–253, Jan. 1995.

[13] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, Nov. 2000.

[14] B. Hendrickson and R. Leland. The Chaco User's Guide: Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, N.M., October 1994.

[15] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[16] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[17] Y. F. Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 10:37–71, 2006.

[18] W. R. Inc. Mathematica edition: Version 7.00. http://www.wolfram.com, 2008.

[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[20] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *Siam Journal on Scientific Computing*, 1998.

[21] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal 49: 291-307.*, 1970.

[22] H. Meyerhenke. Dynamic load balancing for parallel numerical simulations based on repartitioning with disturbed diffusion. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 150–157, 2009.

[23] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel fem simulations by algebraic multigrid. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, 2006.

[24] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *IEEE Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[25] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498, 1996.

[26] A. Sen, H. Deng, and S. Guha. On a graph partitioning problem with applications to vlsi layout. In *Circuits and Systems, 1991., IEEE International Sympoisum on*, pages 2846–2849 vol.5, 1991.

[27] E. Yunis, R. Yokota, and A. Ahmadia. Scalable force directed graph layout algorithms using fast multipole methods. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 180–187, 2012.